

1. 本周工作

1. 阅读《C++ concurrency in action》Chpater 6. Designing lock-based concurrent data structures（内容提炼见附录）；
2. 考试周考试。

2. 下周工作

1. 阅读Chpater 7. Designing lock-free concurrent data structures.
2. 我在10月8日就北大微博数据的VAST2015论文做了组会报告，他们10月14日在自己的博客上发布了论文的中文介绍。之前我不清楚继续在我们的博客上描述他们的论文是否合适，这周我会把博客补上。

附录

Chpater 6. Designing lock-based concurrent data structures

6.1 What does it mean to design for concurrency?

为并发设计数据结构，意味着多个线程可以并发地访问（执行相同或不同的操作）该数据结构，该数据结构不会发生数据丢失、崩溃，不变性将一直保持，因此也没有竞态——这种特性被称为**线程安全（thread-safe）**。

线程安全可以通过mutex实现，保证同一时间只有一个线程可以对数据结构进行操作，这种实现方式被称为**串行化（serialization）**：线程轮流访问被mutex保护的数据，它们事实上是在串行而非并发地访问数据。

一个好的设计，应该更大程度上地实现并发。

6.1.1 Guidelines for designing data structures for concurrency

线程安全：

- 没有线程可以看到别的线程未完成的修改（不变性被打破的状态不被别的线程看到）
- 设计接口时避免竞态产生：提供完整函数的接口，而不是将函数拆分成多个步骤
- 保证发生异常时，不变性不被打破
- 通过限制锁的范围以及尽量避免使用嵌套锁，来减少死锁发生的可能性

为提升并发能力可以问自己的几个问题：

- 是否可以限制锁的范围（6.3.1中的get_head和get_tail）
- 数据不同的部分是否可以由不同的锁来保护（6.3.1中的head和tail）
- 是否所有操作都需要相同级别的保护（6.3.1中的get_head和get_tail）
- 数据结构的简单改变是否可以在不影响语义的同时，增大并发性

6.2 Lock-based concurrent data structures

6.2.1 A thread-safe stack using locks

重现(reproduce)了第三章中引入的threadsafe_stack, 通过分析它的构造函数、析构函数、其它成员函数，说明它为什么是线程安全的。

threadsafe_stack的成员函数pop()调用了用户定义的构造函数或new操作符，导致可能引发死锁。

threadsafe_stack保证了在某一时刻下，最多只有一个线程可以访问它。但考虑一个线程想要从中pop出一个元素，这个线程将要周期性地对它进行pop调用并捕获异常，直至最终获得新的元素——这样是对宝贵CPU时间的浪费。下一小节中的queue采用条件变量，实现了更合理的等待方法。

6.2.2 A thread-safe queue using locks and conditional variables

重现了第四章引入的threadsafe_queue, 主要考察了它用以解决前一小节提出的“等待弹出”问题的wait_and_pop()操作。

指出在线程P1的push()函数中，若data_cond.notify_one()函数所唤醒的线程P2在wait_and_pop()函数中，若抛出异常（如shared_ptr<>对象构造失败），该线程P2将无法及时获得新压入的元素（其他的P3, P4...线程同样无法获得）。对此，有三种解决方法：

- 在push()中，将nofity_one()改为notify_all(), 花销是将导致许多没有不要的唤醒
- 在wait_and_pop()中，若抛出异常，则调用notify_one(), 唤起其它等待的进程
- 在push()中，不再往队列中压入元素，而是压入元素的指针，这将带来两个好处
 - 在wait_and_pop()中，shared_ptr<>的拷贝构造将不会抛出异常
 - 在push()的临界区外，构造新的shared_ptr<>对象，减少了pop()临界区中构造新的shared_ptr<>的动作，增强了并发性。

本小节最后，实现了第三种方法。

6.2.3 A thread-safe queue using fine-grained locks and conditional variables

list 6.4给出了基于list 6.2删减后的单线程queue实现。

ENABLING CONCURRENCY BY SEPARATING DATA

用两个mutex分别保护queue的头尾，通过pop_head()和get_tail()函数，实现了更高的并发能力。

WAITING FOR AN ITEM TO POP

使用std::condition_variable data_cond在fine-grained locks的前提下，实现了：

```
std::shared_ptr<T> try_pop();  
bool try_pop(T& value);  
std::shared_ptr<T> wait_and_pop();  
void wait_and_pop(T& value);
```

6.3 Designing more complex lock-based data structures

6.3.1 Writing a thread-safe lookup table using locks

lookup table需要支持的操作有：

- 增
- 删
- 查
- 改

```
mapped_type get_value(key_type const& key, mapped_type default_value);
```

在执行查询操作（get_value）时，对于不存在的key，返回的value可以有：

- default value, 在该值没有显式提供时，可以使用默认构造函数
- pair<mapped_type, bool>, 第二个参数用以指示key是否存在
- smart pointer, nullptr则代表key不存在

lookup table的实现：

- one mutex: 用一个mutex锁住整张table, 实现6.1中描述的串行化
- shared_mutex: 使用boost中的共享锁，同一时刻允许多个进程进行读操作，但只允许一个进程进行写操作
- 后文将描述的更好的方法

DESIGNING A MAP DATA STRUCTURE FOR FINE-GRAINED LOCKING

实现像lookup table这样的关联容器，可以借助如下的数据结构：

- 二叉树，如红黑树：至少要锁住根节点，即使锁随着访问深度的增加，一起变化，仍不能提升效率
- 排序数组：每次访问需要锁住整个数组
- 哈希表：list 6.11采用哈希表，实现了线程安全的lookup table

table中的每个entry, 都通过list维护一组数据。本小节通过对table的每个entry使用 **shared_mutex**, 增加了程序的并发性。但事实上，通过使用更合理的粒度的锁可以取得更好的并发性。

在下一小节中，我们将使用更合理的粒度的锁实现一个支持迭代操作的链表。

6.3.2 Writing a thread-safe list using locks

6.3.1中的list需要支持如下操作：

- 增
- 删
- 查
- 改
- 复制

本小节中，通过对list中的每个节点（而不是上一节中，table的每个entry对应的list）加锁，实现了更高的并发性。